Snake Game AI: Searching Problem and Deep Q-Learning

Yi Lin
y477lin@uwaterloo.ca

Qingze Ren
q25ren@uwaterloo.ca

Wensen Li
h552li@uwaterloo.ca

Abstract

We study the classic, single player video game, the Snake Game using AI method learnt in this course. We first formulate the Snake Game as a searching problem and uses A* search algorithm to play the game. On the other hand, we formulate it as a reinforcement learning problem, and use a Deep Q-learning (DQL) based agent, a combination of active Q-learning and deep neural networks, to beat the game. Implementation codes are provided, and simulation results are shown. We observe that A* search gives an impressive performance in terms of score, while DQL agent beats A* search in terms of running time significantly. Future study approaches are made according to the results.

1 Introduction

Playing video games is one of my main entertainment. Recently, using artificial intelligence method to play video games has become a challenge in the field. One must have heard Alpha GO, an AI designed to beat the world's greatest GO players Fan Hui in 2015, and Lee Sedol in 2016. Later in 2018, AlphaStar, another AI program designed to play StarCraft II, defeated a world champion Grzegorz "MaNa" Komincz with 5 games in a role. This model is considered to be a milestone since [4] suggests that this game is "real-time play, partial observability, no single dominant strategy, complex rules that make it hard to build a fast forward model, and a particularly large and varied action space."

Therefore we found it exciting to choose an AI-trained gaming topic for this project. While highly competitive games between multiple agents (like GO and StarCraft II mentioned above) are difficult to implement, we decided to switch our focus towards more classic single-player games, which is also a popular research area in AI gaming. Designing algorithms for this type of game helps players to find a more efficient way beating the game by themselves. In this project, we develop and train an AI agent to play the classic Snake Game with both reinforcement learning and deep learning method.

The original form of the game is very easy to understand: In a $n \times n$ square board, the player controls a "snake" in the form of a non-straight line, with one end of the line being the "head", and the rest of the line being the "tail". It has 3 possible moves: go straight, turn left or turn right (relative to the forward direction). The objective is to control the snake and eat as many fruit as possible that will appear on the board randomly. There is only one fruit on the board at a time. Each time the snake consumes a fruit, its length is increased by one at the tail. The game is over when the snake's head collides either into the wall or with itself.

There has been a number of paper digging into different kinds of algorithm that can solve or learn to play the snake game. We will discuss the details in section 2 of this proposal. 2 criteria are considered when evaluating the algorithms: max score and moves taken. There are brute force algorithms that guaranteed to find an optimal solution in terms of score to this game, but usually takes unacceptable number of moves when n is very large (e.g. 20). On the other hand, searching algorithms are able to

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

find the shortest path to the fruits every time, but can fail with high probability. Finally, learning and genetic algorithms are designed to beat the game with maximum score and using minimum moves.

We propose to build 2 AI models for the Snake game. First, we will implement the A* search algorithm to learn the Snake game as a searching problem. Secondly, we use decision theory to model the game, and combining Q-learning and deep neural network (will be called Deep Q-Learning or DQL) to train an agent. Q-learning learns a Q-value table by generating experiences, and determines the policy based on Q-values which depends on actions and states. DQL uses a deep network to approximate the Q-values instead. We also make an strong assumption that the snake has a sensor, meaning it is able to detect the relative position of all kinds of object (fruit, wall, body) on the game board. This assumption allows us to simplify the formulation of the snake game, and avoid using complicated neural networks. We have provided the code in the LEARN dropbox.

This report is ordering in the following way. Section 2 discuss

2 Related Work

Previous researches regarding the snake game can be divided into two categories based on approaches: non-machine-learning (non-ML) and machine learning. We will discuss them separately.

2.1 Non-ML Approach

To start with, the snake game has a brute force, trivial solution: the Hamiltonian cycle approach. For a $n \times n$ square board regardless the choice of n, we can always find a a cyclic path that goes through every square on the board without crossing itself, also known as the Hamiltonian Cycle. Following this path allows the snake to travel the whole board multiple times, with 1 fruit guaranteed each cycle, until the length of the snake is equal to n^2.

[1] showed the complexity of the Hamiltonian Cycle approach for the snake game to be $\Theta(n^4)$. For example, on a 20 by 20 board, it will take on average about 40,000 moves to achieve maximum score. This number is clearly beyond our expectation. Interestingly, [1] also discussed a deeper relationship between snake game, Hamiltonian cycle and Markov Decision Process that helps solve non-deterministic NP-hard problem in an efficient way.

Searching is a classic way in AI of finding shortest path from current state to the target state, which clearly can be applied to the snake game as well. [2] gives a comprehensive performance analysis of numerous searching algorithms on the snake game, including both uninformed and informed search. With multiple games simulated by different searching methods, Appaji concluded that A^* search gives the best performance in terms of both scores and number of moves. All the searching algorithms beat Human Agent, which randomly assign moves.

2.2 ML Approach

Genetic algorithms are another popular approach to this type of problem. This approach is modeled off of biological evolution and natural selection. This type of machine learning model maps perceptual inputs to action outputs. To start, random models are created and simulated. The best one is picked with highest value from a fitness function and made random mutations. By repeating over time, the evolutionary pressure will select for better and better models.

[3] gives a detailed implementation of the traditional genetic algorithm with fitness function $moves^2 \times 2^{score}$. Result showed a fairly stable score-to-generation curve with slight fluctuation. [5] provides a more novel approach. It defines 4 rating functions considering smoothness, space, and food, and uses evolution algorithm to optimize their weights through generation. Although different AI controllers trained on this model can play better in a given game, they cannot play well in unknown games.

Reinforcement learning (RL) is fast-growing research field in AI. At a very basic level, RL involves an agent, an environment, a set of actions that the agent can take, and a reward function that rewards the agent for good actions or punishes the agent for bad actions. As the agent explores the environment, it updates its parameters to maximize its own expected reward. Deep Reinforcement learning (DRL), as the name suggests, is a combination of RL and deep neural network. DRL enables RL to handle problems with high-dimensional state and action spaces [6], [7].

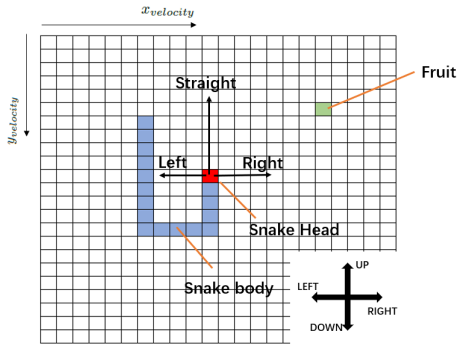

Figure 1: A Snake Game board consists of the snake's head (red), its body (blue) and a fruit (green). Possible actions are: go straight (keep current direction), turn left or turn right. These directions are relative to the snake's current direction. It can also be relative to the board, i.e. either up, right, down or left.

The combination of the Q-Learning algorithm [8] and deep neural networks has given rise to the Deep Q-Learning (DQL). [9] first applied DQL to the snake game in 2021. Its proposed agent uses sensor data measurements to avoid complicate convolutions neural networks. Also, a particular attention has been given to the neural network hyperparameters tuning process, which influences the level of the achieved performance.

3 Problem Formulation

In this section, we will discuss how we formulate the Snake Game formally as a searching problem and as a Markov Decision Process, and how the DQL is defined in the latter setting.

First let's revisit the rule of Snake game: Each game starts with a Snake with length 1, i.e. only the head. During the game a fruit will spawn in a random position on the game board. The snake can consume the fruit by entering the grid that contains the fruit by its head. Note that there can only be 1 fruit at a time on the game board. Whenever a fruit is consumed, the length of the snake increases by 1, and there will be a new one spawned in a new random position. The game is over when the snake's head collides either with the wall or with its own body.

3.1 Search Problem and A* algorithm

Given a $n \times n$ board, a state includes the current position of the position of snake's head and all the positions of snake's body. As a result in the initial state, the position of snake's body is an empty set. There is no ultimate goal state, but the goal states are consecutive. For a single time instance, there is only one food on the board, and the current goal state is the position of the fruit. When the fruit is consumed (goal state is reached), a new fruit (a new goal state) will be defined. The neighbor of each state has at most 3 elements, either the snake's head moves in the current direction, to the left or to the right. The head will proceed to one of the 3 neighbour grid, and the position of snake's body will be updated accordingly. There are two restrictions of the neighbour: 1. the target grid cannot be the snake's body; 2. the target grid cannot be the wall. If all 3 neighbouring grid do not pass the restrictions, then the searching ends and game is over.

The cost of a state to the goal state is defined to be the Manhattan distance. In order to use A* search, we need a heuristic function which is defined as the Euclidean distance. It is easy to verify that the heuristic funtion is both admissible and consistent.

3.2 RL Terminology

We model the snake game as a Markov Decision Process $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \gamma \rangle$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the action space, $\mathcal{R}(s) \ \ \forall s \in \mathcal{S}, a \in \mathcal{A}$ is the reward function, and γ is the discount factor. Note that we an MDP is modelled in a stochastic manner, i.e. a transition probability distribution should be defined, but in the Snake Game setting we model it as deterministic. For each state s, an action

will be assigned, and the agent will follow the action with full probability. This assignment is refer as policy $\pi(\cdot) := \mathcal{S} \to \mathcal{A}$. In each time stamp $t = 1, 2, \ldots$, the agent receives a representation of environment state $s_t \in \mathcal{S}$, follows the action given by the policy $a_t = \pi(s_t) \in \mathcal{A}$, entering a new state s_{t+1} and receives a reward $r_t = \mathcal{R}(s_{t+1})$.

The agent's objective is to find the optimal policy, i.e. the policy that maximizes the expected utility. Same in class, we define $Q^\pi(s, a)$ as the Q-value function given a policy π (same in class, the expected value of the total discounted reward). The optimal policy π^* maximizes Q^π and satisfies the Bellman's equation:

$$Q^*(s, a) = R(s) + \gamma \sum_{s'} P\left(s' \mid s, a\right) \max_{a'} Q\left(s', a'\right). \tag{1}$$

Of course in the Snake Game setting we can ignore the transition probability $P\left(s' \mid s, a\right)$. Using temporal difference, we can approximate $Q^*(s, a)$ by the following update rule:

$$Q\left(s_t, \mid a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha \left[r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q\left(s_{t+1}, a'\right) - Q\left(s_t, a_t\right)\right] \tag{2}$$

3.3 Snake Game RL system

Next we will formulate the Snake Game as a reinforcement learning system. To begin with, the environment consists of 3 element: the game board, the snake, and the fruit. The game board is a $n \times n$ square grid. Each cell is occupied by either: nothing, a fruit, a snake's head, or a part of snake's body of length 1. We use coordinate (x, y) to locate the position of the fruit and the snake, where x is the horizontal axis and y is the vertical axis (downwards)See Fig 1 for an example of the game.

The reward system is the following. The agent receives a reward of 10 when consuming a fruit. The game is over when the snake's head collides either with the wall or with its own body, and the total reward is decreased by 10. All other situations has 0 reward. Note that the reward function is used to train the DQ network described in section 3.4. On the other hand, we use a concept of "game points counter" to measure the performance of the agent. The counter starts at 0 at the beginning of each game. When the agent consumes a fruit, the game points counter is increased by 1. When the game is over, the counter is reset to 0.

Assuming that the snake has a sensor (is able to detect the relative position of all kinds of objects on the board), a state $s \in \mathcal{S}$ is defined by 11 boolean variable as follows:

$$s = [D_{straight}, D_{left}, D_{right}, F_{left}, F_{right}, F_{up}, F_{down}, M_{left}, M_{right}, M_{up}, M_{down}]$$

The first 3 entries $D_{straight}, D_{left}, D_{right}$ indicates if there is a dangerous position in the straight, left and right direction relative to the snake's head. A dangerous position is either the wall or the snake's body. Examples are shown in Fig 2. The next 4 entries $F_{left}, F_{right}, F_{up}, F_{down}$ indicates the relative position of the fruit to the snake's head. An example is shown in Fig 3.

The last 4 elements show how the snake moves. Let's first define the action space. In the Snake Game, $\mathcal{A} = \{Straight, Left, Right\}$, i.e. the snake can move either straight, left or right **relative to the head**, causing the snake's head to move either up, right, down or left **relative to the board**. Mathematically, suppose at time t, the snake's head is positioned at $(x(t), y(t))$. After the agent picks an action, $x_{velocity}$, $y_{velocity}$ and M values will be specified accordingly:

- If the snake moves to the right (relative to the board), then $x_{velocity} = 1$, and $M_{right} = 1$
- If the snake moves to the left (relative to the board), then $x_{velocity} = -1$, and $M_{left} = 1$
- If the snake moves upwards (relative to the board), then $y_{velocity} = -1$, and $M_{up} = 1$
- If the snake moves downwards (relative to the board), then $y_{velocity} = -1$, and $M_{down} = 1$

In other words, the action selected at time t affects the state entries that are related to the movement. Moreover,

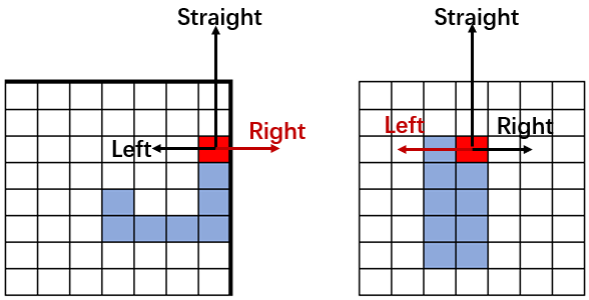

Figure 2: 2 examples of dangerous position. In the first figure, the wall (black bold border) is towards the right of the snakes head (red square), therefore $D_{right} = 1, D_{left} = 0, D_{straight} = 0$. In the second figure, the snake's body (blue square) is towards the left of snake's head, $D_{right} = 0, D_{left} = 1, D_{straight} = 0$

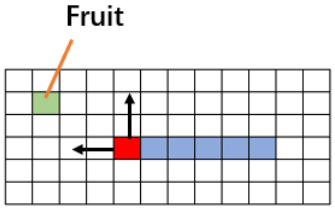

Figure 3: An example of the fruit position. Here, fruit is towards the left and up relative to snake's head. Therefore, $F_{left} = 1, F_{right} = 0, F_{up} = 1, F_{down} = 0$

$$x(t+1) = x(t) + x_{velocity}$$
$$y(t+1) = y(t) + y_{velocity}$$

3.4 Deep Q-Learning Network Based Agent

In this section we introduce the Neural Network (DQN) architecture, and present the algorithm of train the agent. To clarify, DQN is a representation of policy $\pi(a|s)$ that determines the best action at each state. Therefore, DQN is another form of Q-value table that uses neural network to approximate Q-values and make a decision. For a complete Snake Game, the agent will make numbers of decisions on the action taken at each state, and the environment will provide rewards. The DQN will be trained on a set of states, actions, rewards to learn the actions that maximizes the expected total reward from each state.

The architecture of DQN varies with game, but typical pixel games like the Snake Game usually takes the whole image pixels (the whole game boards at each state represented as image pixels) as inputs to the DQN. This requires the neural network's ability to process images. Complicated deep CNN is one example of such architecture. However in this project, we assumed that the snake has a sensor ability, which allows the agent to detect the relative position of any objects on the game board, including its own body, the fruit and walls. Such assumption allows us to keep track of these position information within the state definition (see Section 3.3). Therefore we don't actually need complicated neural net architectures.

For this project, we will use a simple 2-layer feed-forward network (tunable). The input layer has 11 nodes, which is consistent to the size of sate vector s. Then the hidden layer consists of a customized number (e.g. 256) of units, followed by a ReLU layer. The output layer has 3 nodes, representing the predicted Q-values for the 3 possible actions: go straight, turn left or turn right. The agent will then use a maximum function to choose the action with the highest Q-value.

n	Score	Run Time
5	avg 12.4	<1s
10	avg 17.6	<1s
15	avg 29.8	<1s
20	avg 36	1s
50	123	3s
	141	8s
	170	13s
100	183	26s
	264	64s
	331	145s
300	NA	>20min
600	NA	>1h

Table 1: Score and runtime on different size of Snake game (n) using A* search algorithm. "avg" stands for average score. For $n = 300,\ 600$, it took too long to finish (game did not end). Therefore the score is not presented.

The reason why we only use 1 hidden layer is because of the Universal Approximation Theorem. We are essentially

The algorithm to train the agent is given by the following:

1. At the start of the training process, Q-values $Q(s, a)$ are initialized randomly.
2. For each epoch (each game):
   (a) The agent selects an action based on the exploration-exploitation method (active Q-learning). When the epoch is less than a fixed number N_e, the agent explores the environment with probability ϵ.
   (b) Based on the action selected, the agent generates an experience $\langle s_t, a_t, s'_{t+1}, r_t \rangle$. These values are stored as a single data point.
   (c) Repeat (a) and (b) until the game is over.
3. After each epoch (each game), the proposed neural network is trained using a subset of the data collected during step 2(b).

Training a neural network requires both the predicted value and the ground truth to calculate the loss function. In this case, the expected ground truth will be calculated iteratively using equation (2) with the data collected. The predicted values are also obtained using the network model with the data collected. Another way is to update the Q-values with the game's progress instead of calculating it after each game. We pick ADAM as the optimizer and mean squared error (MSE) as loss function to train the network.

After the training phase, we can use the neural network to play new games, each time selecting the action that maximizes the output of the neural net.

4 Experiment and Result

4.1 A* Search Results

Previous work [2] concluded that A* search beats all the other searching algorithms and human agent (random moves). To make compare with the performance of a DQN-based agent, we also implemented A* search. We chose the Euclidean distance between the snake's head and the fruit as the heuristic function, and records the number of fruits eaten before game over. In order to test the scalability of A* search, different game board size n is chosen. The results are in table 4.1.

Several observations can be made:

1. For all the observable results (excluding NAs), the snake's length achieved a very large value. For example, when $n = 100$, the max score we achieved is 331, which means the

snake's size is 3 times of the board length! This is already impressive since when the snake is long enough, A* search can still finds a valid path to the fruit without bumping into itself.

2. For $n \leq 20$, the running time is fast. However we learnt from class that A* search running time scales exponentially with the board size. For example, when $n = 100$, the running time is more than twice when the score increases by around 80. The game is not able to finish within 10 minutes if $n = 300$, or within an hour if $n = 600$.
3. Our finding agrees with previous works on this topic: the A* search algorithm performs well on small game sizes, but does not scale well in terms of the running time. To improve the running time, one have to find more effective methods of reducing the searching space, since there are way too many moving options in the Snake Game.

4.2 Deap Q-learning Results

For DQL, we fixed the size of the game board $n = 640$. This is a larger number than what we used in A* search, since it is expected that Deep learning method runs faster on larger instances. The following hyperparameters can be tuned in this model:

- ϵ: The exploration-exploitation trad-off, the probability of exploration.
- N_e: Number of maximum exploration games.
- Learning rate α in equation (2)
- Architecture of the feed-forward network.
- Batch size: the size of the subset that is used for Q-value update after each game (step 3 in the algorithm).

For time limitation, we did not have a chance to tune these hyperparameters systematically. Table 4.2 shows the simulation results, from which we make the following observations:

1. The running time of a single game is significantly faster than A* search algorithm. In previous result it took more than one hour to finish one game using A* search on a $n = 640$ board, but it only took DQL a few seconds.
2. As a direct result of the previous observation, in a fixed time period, DQL is able to achieve much higher score than A* search. This agrees with several properties of deep learning that we studied in class. However, for the DQL algorithm the agent tends to end games earlier than expected!
3. The average training average increases with number of epoch (games) trained on. This is usually a general trend in deep learning, but it also has a risk of overfitting (which is not defined and tested in this project). Note that the average training error includes the games from exploration stage (see Fig 4). This is the reason why average score is way less than the best score.
4. The agent does not like to take too many exploration steps ($N_e = 50$ when epochs is 150). The agent also does not like too small learning rate (e.g. 0.00005) when the number of learning games is 300. Both adjustments lowers the score.
5. From Fig 4, we can see that the score at the exploration stage (in this case, the first 80 games) is very low (below 3). When the exploitation stage starts, the score immediately increases to a very high level, and continuous to alternate between low and high scores, leading to a very unstable learning curve even when the epoch number is high. We are very curious about how to explain this phenomenon, yet this learning curve highly agrees with the numerous simulations that appeared in [9].

Despite the time improvement, it seems that the model is not very well trained since the best score has only reached 59. This means the agent tend to end games earlier than expected, with no clear reasons why this happens. There are several improvements that can be made for model tuning in the future.

- Adjust the reward system. The current reward for ending a game is too low given the results

epochs	ϵ	N_e	LR α	Batch size	Training Avg	Best Score	Training time	Run time per game
300	0.25	40	0.001	800	16.14	**59**	around 40 min	around 8s
300	0.25	40	0.005	800	15.03	53		
300	0.5	100	0.001	800	12.04	45		
300	0.35	70	0.00005	800	6.893	48		
150	0.5	50	0.0001	800	0.728	7	12 min	around 4.8s
600	0.4	80	0.001	800	**18.17**	58	120 min	around 12s

Table 2: Results of DQN simulations. Best scores are marked bold.

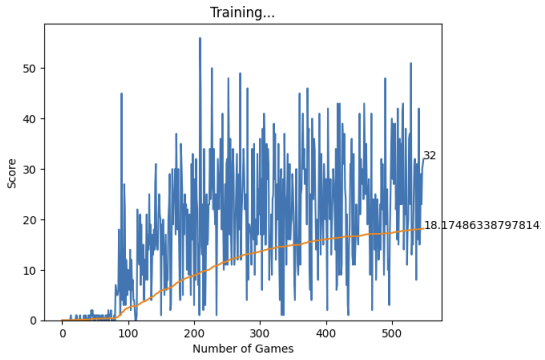

Figure 4: The learning curve(blue): score v.s. number of game trained from the highest score attempt (last row of Table 4.2. The yellow curve is the moving average.

- Try more options for neural network architecture. Based on the assumptions we made for the "sensor" ability of the agent, we only tried a very straightforward architecture: the feed-forward network. More options can be tested, such as using the convolutional neural network (CNN) on the whole game board image pixel. CNNs are designed to better extract image information from images than other neural networks, and thus has probably more potential than the current model.
- Expanding the batch size. Due to memory issue concerns, we only selected 800 instances from the data collection each time we train the network, which potentially has randomness issues. Not all informative moves are selected.
- Explore the influence of dropout layer. Dropout is a common technique in deep learning to combat overfitting by relaxing the complexity of the model. We expect that by implementing dropout, the agent would have a different behavior.
- Tune hyperparameters in a more systematic way to examine the effective of each.

5 Conclusion

In this project, we attempted to use AI methods to beat the classic single player game, the Snake Game. We formulate the game as: 1. a searching problem and uses A* search to solve it; 2. a reinforcement learning problem and uses Deep Q-learning (using deep neural networks to approximate Q-values) to beat the game. We implemented both algorithms and analyzed their performance. We found that A* is able to given a very impressive performance if given enough time, but scales poorly when the size of the board. On the other hand, Deep Q-learning based agent can observe, decide and execute the best action in a very short time within seconds.

However, the simulation result showed strangely low performance on the DQL agent that bumps into walls and body too early in a large board, meaning the model is not well-trained, or the RL system is not well-defined. Several suggestions of future studies are made. Nonetheless, we believe this project is a very good realization of important course topics on a real-life, complicated enough scene. We have established a deeper understanding on A* searching, active Q-learning, and deep neural networks.

References

[1] Du, K., Gemp, I., Wu, Y., & Wu, Y. (2022). AlphaSnake: Policy Iteration on a Nondeterministic NP-hard Markov Decision Process. *arXiv preprint arXiv:2211.09622.*

[2] Appaji, N. S. D. (2020). Comparison of Searching Algorithms in AI Against Human Agent in the Snake Game.

[3] Białas, P. Implementation of artificial intelligence in Snake game using genetic algorithm and neural networks.

[4] Arulkumaran, K., Cully, A., & Togelius, J. (2019, July). Alphastar: An evolutionary computation perspective. In *Proceedings of the genetic and evolutionary computation conference companion* (pp. 314-315).

[5] Yeh, J. F., Su, P. H., Huang, S. H., & Chiang, T. C. (2016, November). Snake game AI: Movement rating functions and evolutionary algorithm-based optimization. In *2016 Conference on Technologies and Applications of Artificial Intelligence (TAAI)* (pp. 256-261). IEEE.

[6] K. Arulkumaran, M. P. Deisenroth, M. Brundage & A. A. Bharath, "Deep reinforcement learning: A brief survey", *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26-38, 2017.

[7] D. P. Bertsekas, "Feature-based aggregation and deep reinforcement learning: A survey and some new implementations", *IEEE/CAA Journal of Automatica Sinica*, vol. 6, no. 1, pp. 1-31, 2018.

[8] D. P. Bertsekas, Dynamic programming and optimal control, MA:Athena scientific Belmont, vol. II, 2012.

[9] Sebastianelli, A., Tipaldi, M., Ullo, S. L., & Glielmo, L. (2021, June). A deep q-learning based approach applied to the snake game. In *2021 29th Mediterranean Conference on Control and Automation (MED)* (pp. 348-353). IEEE.

[10] Almalki, A. J., & Wocjan, P. (2019, December). Exploration of reinforcement learning to play snake game. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)* (pp. 377-381). IEEE.